DAM
Desarrollo de Aplicaciones Multiplataforma
2º Curso

AD
Acceso a Datos

UD 8
Programación de componentes
de acceso a datos
Parte 3

IES BALMIS
Dpto Informática
Curso 2019-2020
Versión 1 (03/2019)

UD8 – Programación de componentes de acceso a datos

ÍNDICE

9. Uso de API Rest

API (Application Programming Interface o Interfaz de Programación de Aplicaciones) es un conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

El término REST (Representational State Transfer) se originó en el año 2000, descrito en la tesis de Roy Fielding, padre de la especificación HTTP.

REST es un modelo de arquitectura web basado en el protocolo HTTP para mejorar las comunicaciones cliente-servidor que utiliza un conjunto de restricciones con las que podemos crear un estilo de arquitectura software.

API Rest o Restful es un servicio web que implementa la arquitectura REST y que responde a diferentes peticiones utilizando generalmente un intercambio de datos en formato JSON o XML

Algunos frameworks con los que podremos implementar nuestras API Rest son:

- JAX-RS y Spring Boot para Java,
- Django REST framework para Python,
- Laravel para PHP o
- Restify para Node.js

Hoy en día la mayoría de las empresas utilizan API REST para crear servicios. Esto se debe a que es un estándar lógico y eficiente para la creación de servicios web.

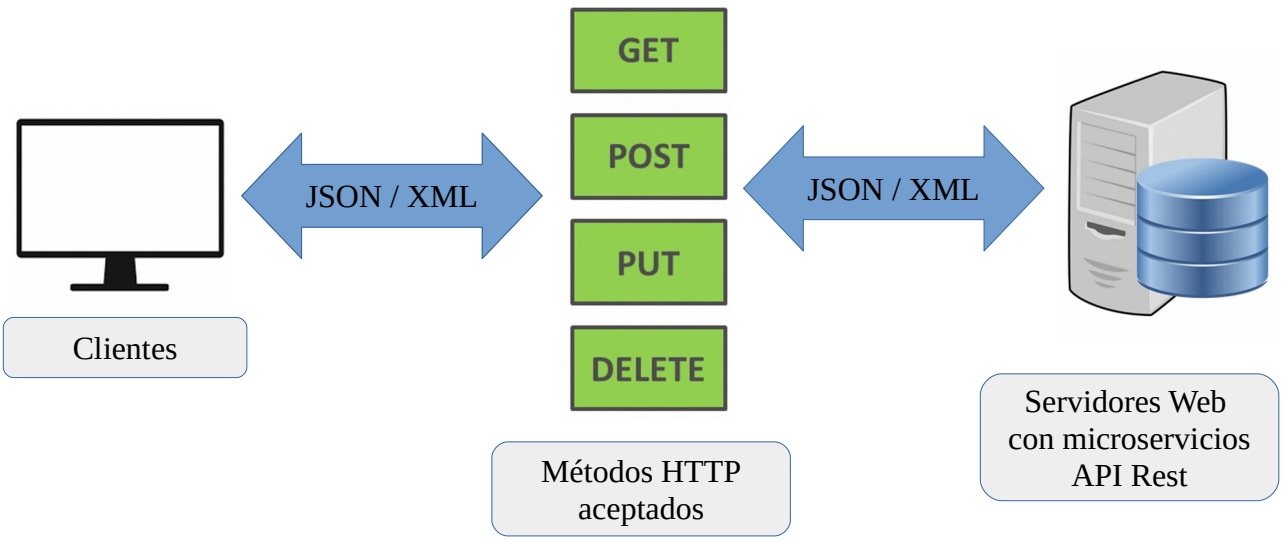
Por poner algún ejemplo tenemos los sistemas de identificación de Facebook, la autenticación en los servicios de Google (hojas de cálculo, Google Analytics, …).

El concepto de **Open Data (Datos abiertos)** es una filosofía y práctica que persigue que determinados tipos de datos estén disponibles de forma libre para todo el mundo, sin restricciones de derechos de autor, de patentes o de otros mecanismos de control. Tiene una ética similar a otros movimientos y comunidades abiertos, como el software libre, el código abierto (open source) y el acceso libre (open access).

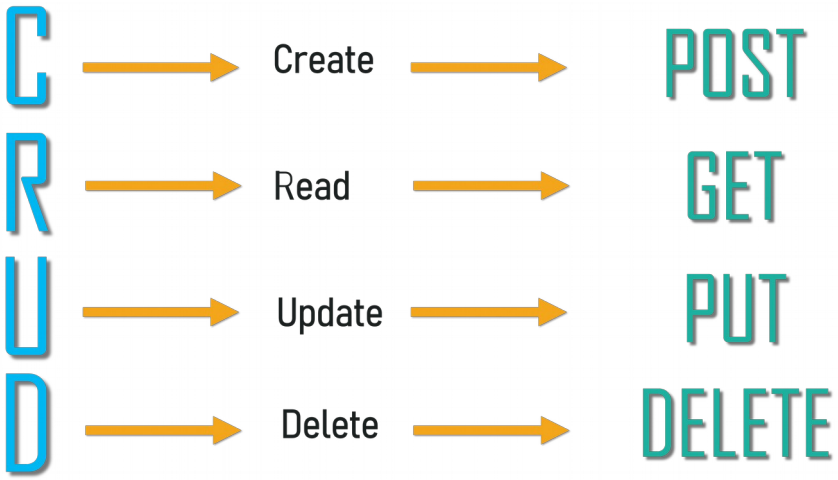
Hay muchos servicios de Open Data y la mayoría se ofrecen a través de servicios API Rest. Algunos ejemplos importantes:

- **OpenStreetMap** => https://wiki.openstreetmap.org/wiki/API
  - Ejemplo: https://www.openstreetmap.org/api/0.6/way/81131976
- **Aemet OpenData** => https://opendata.aemet.es/centrodedescargas/inicio
  - Especificación de métodos:
    - https://opendata.aemet.es/AEMET_OpenData_specification.json
  - Ejemplo de link en Aemet con JSON:
    - https://opendata.aemet.es/opendata/api/maestro/municipios/?api_key=XXX

Las operaciones más importantes que nos permitirán manipular los recursos son:.



Cada método se usa para realizar una acción en la base de datos:



Aquitectura API Rest
https://juanda.gitbooks.io/webapps/content/api/arquitectura-api-rest.html

9.1 Probar el funcionamiento de un servicio API Rest

Entraremos en la URL Base del API Rest de ejemplo:

http://www.riconet.es/fp/apirest

Probar con el navegador el método GET

Sin instalar ningún plugin en el navegador, podemos probar el método **GET**. Basta con introducir la URL de los ejemplos:

http://www.riconet.es/fp/apirest/datos
http://www.riconet.es/fp/apirest/libros
http://www.riconet.es/fp/apirest/libros/3
http://www.riconet.es/fp/apirest/libros/count

El formato obtenido por defecto es HTML para la primera y JSON para el resto.

Probar API Rest con Postman

Para probar un API Rest, lo ideal es utilizar una aplicación específica que nos permita parametrizar todos los datos necesarios y **Postman** es una de las más utilizadas.

Postman – Web oficial
https://www.getpostman.com/downloads/

Generalmente la información que vamos a utilizar en el envío son:

Método	Aunque hay más, se suelen usar GET, POST, PUT y DELETE
URL Base	Es la parte inicial de la URL de tipo http que contiene hasta la carpeta donde instalamos nuestro API Rest
URL RewriteRule	Es la parte final de la URL que se reescribe y procesa para procesar la petición (REQUEST). A este proceso también se le llama "URL Amigable".
Datos de QUERY	Es la parte de la URL posterior al carácter **`'?'`** donde se añaden parejas de **parámetro=valor** separadas por el carácter **`'&'`**
Content-Type	Es el tipo de los datos de envío. En API Rest se suele usar json o xml y se indica con el valor "**application/json**" y "**application/xml**" respectivamente
Datos de envío	Es una cadena de caracteres en el formato indicado por "**Content-Type**"
Accept	Es el formato en el se desea que el servidor nos devuelva los datos

La respuesta contendrá al menos la siguiente información:

Datos de respuesta	Es una cadena de caracteres en el formato indicado por "**Accept**" en la petición (REQUEST)
Response	Código de estado de la respuesta (**HTTP Status Code**). Es un número entero de 3 dígitos donde el primer dígito del Código de estado define la clase de respuesta y los dos últimos dígitos el tipo de respuesta https://www.tutorialspoint.com/http/http_status_codes.htm

Veamos el formato general:

`http://dominio/pathApiRest/rewriteRule?queryString`

Probaremos algunos ejemplos con Postman para comprobar el funcionamiento:

Obtener en JSON todos los registros de la tabla libros	
Método	`GET`
URL	`http://riconet.es/fp/apirest/libros`
Headers => Content-Type	
Datos de envío	
Headers => Accept	`application/json`

Obtener en XML el registro con id=3 de la tabla libros	
Método	`GET`
URL	`http://riconet.es/fp/apirest/libros/3`
Headers => Content-Type	
Datos de envío	
Headers => Accept	`application/xml`

Insertar en la tabla libros el equivalente a:
`INSERT INTO libros (id, titulo, autor)`
`VALUES (101, 'El instituto', 'Stephen King');`
enviando los datos en JSON y recibiendo la respuesta en XML	

Método	`POST`
URL	`http://riconet.es/fp/apirest/libros`
Headers => Content-Type	`application/json`

Datos de envío	`Body => Raw => JSON`
	`{`
	`"id":101,`
	`"titulo":"El instituto",`
	`"autor":"Stephen King"`
	`}`

Headers => Accept	`application/xml`

Se puede consultar la información de especificación completa del API Rest de ejemplo pulsando en el botón de "**Descargar PDF**":

http://www.riconet.es/fp/apirest

9.2 Utilizar un servicio API Rest desde Java

Para poder usar un servicio de API Rest desde Java con Netbeans tendremos que usar dos librerías:

`JAX-RS 2.0`	Es la librería principal que nos permite conectar con el servicio de API Rest
`Jersey 2.5.1 (Jax-RS RI)`	Es una librería necesaria (dependency) para que funcione JAX-RS

Mediante un ClientBuilder de JAX-RS podremos conectar y obtener respuesta (REQUEST) de un servicio API Rest.

API Rest – Clases
https://javaee.github.io/javaee-spec/javadocs/javax/ws/rs/client/Client.html https://javaee.github.io/javaee-spec/javadocs/javax/ws/rs/client/WebTarget.html https://javaee.github.io/javaee-spec/javadocs/javax/ws/rs/client/Invocation.Builder.html https://javaee.github.io/javaee-spec/javadocs/javax/ws/rs/core/Response.html

Probaremos a crear un pequeño proyecto que conecte con nuestro ejemplo de API Rest para obtener los datos en JSON de libro con id='4'.

ApiRest01Get

```
// Objetos para realizar la petición
Client client = ClientBuilder.newClient();

WebTarget webTargetBase;
WebTarget webTargetSolicitud;
Invocation.Builder invbuilder;
Response response;

// Indicar la URL del API Rest
webTargetBase       = client.target("http://localhost/fp/apirest");
webTargetSolicitud  = webTargetBase.path("/libros/4");

// Aportar los formatos y datos a la llamada
invbuilder = webTargetSolicitud.request();
invbuilder.header("Content-Type", "application/json");
invbuilder.accept(MediaType.APPLICATION_JSON);

// Ejecutar el método
response = invbuilder.get();
```

Los métodos ejecutados son:

target	Permite asignar la URL Base del servicio API Rest
path	Es la parte de la URL que permite parametrizar el servicio requerido
request	Crea un objecto para invocar el método
header	Asigna variables de cabecera como "Content-Type" para indicar el formato de los datos enviados
accept	Indicar el formato de los datos que vamos a recibir
get	Invoca el método

Una vez ejecutado obtenemos un objecto Response que nos proporciona es código de estado (response status code) y los datos de la respuesta.

Los códigos de estado están tipificados en la especificación de API Rest

API Rest – Status Code
https://restfulapi.net/http-status-codes/

Para mostrar el resultado probaremos con el siguiente código:

```
// Mostrar la respuesta obtenida
System.out.println("Status:");
System.out.println("=======");
System.out.println(response.getStatus());
System.out.println();

System.out.println("Data Body:");
System.out.println("==========");
System.out.println(response.readEntity(String.class));
System.out.println();
```

Para crear un código más simple, podemos encadenar los métodos hasta obtener el objeto **response**:

ApiRest02Get

```
// Objetos para realizar la petición
Client client = ClientBuilder.newClient();

Response response = client
              .target("http://localhost/fp/apirest")
              .path("/libros/4")
              .request()
              .header("Content-Type", "application/json")
              .accept(MediaType.APPLICATION_JSON)
              .get();

// Mostrar la respuesta obtenida
System.out.println("Status:");
System.out.println("=======");
System.out.println(response.getStatus());
System.out.println();

System.out.println("Data Body:");
System.out.println("==========");
System.out.println(response.readEntity(String.class));
System.out.println();
```

Para recibir los datos en XML bastará con cambiar el parámetro en el método **accept**:

ApiRest03Get

```
...
Response response = client
              .target("http://localhost/fp/apirest")
              .path("/libros/4")
              .request()
              .header("Content-Type", "application/json")
              .accept(MediaType.APPLICATION_XML)
              .get();
...
```

Si queremos añadir un nuevo libro, necesitaremos enviar los datos del nuevo libro al servidor.

Como hemos visto anteriormente con Postman, podemos enviar los datos en varios formatos:

- raw => JSON (application/json)
- raw => XML (application/xml)
- raw => TEXT (x-www-form-urlencoded)

Para nuestro proyecto de insertar un libro, calcularemos el ID obteniendo el número de libros y le sumándole uno. El resto de campos los pedimos al usuario.

ApiRest04Post

```
// DECLARACIÓN DE VARIABLES
String body;
JsonReader jsonReader;
JsonObject json;

int id;
String titulo;
String autor;

Scanner teclado = new Scanner(System.in);

Response responseCount;
Response responsePost;

// PROCESO DE POST
Client client = ClientBuilder.newClient();

responseCount = client.target("http://localhost/fp/apirest")
                      .path("/libros/count")
                      .request()
                      .header("Content-Type", "application/json")
                      .accept(MediaType.APPLICATION_JSON)
                      .get();
if (responseCount.getStatus() == 200){
  body = responseCount.readEntity(String.class);
  // System.out.println("Response: " + body);

  jsonReader = Json.createReader(new StringReader(body));
  json = jsonReader.readObject();

  // ***************
  // DATOS DEL LIBRO
  id = Integer.valueOf(json.getString("total_registros"));
  id = id + 1;
  System.out.println("Id Libro: "+id);

  System.out.print("Introduzca título: ");
  titulo = teclado.nextLine();

  System.out.print("Introduzca autor: ");
  autor = teclado.nextLine();
```

```
  // ***************
  // JSON DEL LIBRO
  String libro = "{ 'id': "+id+", 'titulo':'"+titulo+"',"+
                                  "'autor':'"+autor+"' }";
  libro = libro.replaceAll("'", "\"");
  System.out.println(libro);

  // ***************
  // POST DEL LIBRO
  responsePost = client.target("http://localhost/fp/apirest")
                       .path("/libros/")
                       .request()
                       .header("Content-Type", "application/json")
                       .accept(MediaType.APPLICATION_JSON)
                       .post(Entity.json(libro));
  if (responsePost.getStatus() == 201){
    body = responsePost.readEntity(String.class);
    // System.out.println("Response: " + body);

    jsonReader = Json.createReader(new StringReader(body));
    json = jsonReader.readObject();
    System.out.println(json.getString("mensaje"));
  } else {
    body = responsePost.readEntity(String.class);
    // System.out.println("Response: " + body);

    jsonReader = Json.createReader(new StringReader(body));
    json = jsonReader.readObject();
    if (json.containsKey("mensaje")) {
      System.out.println("ERROR: "+json.getString("mensaje"));
    }
    if (json.containsKey("sqlError")) {
      System.out.println(json.getString("sqlError"));
    }
  }

} else { //  responseCount
  body = responseCount.readEntity(String.class);
  // System.out.println("Response: " + body);

  jsonReader = Json.createReader(new StringReader(body));
  json = jsonReader.readObject();
  if (json.containsKey("mensaje")) {
    System.out.println("ERROR: "+json.getString("mensaje"));
  }
}
```

Si los datos los queremos enviar en formato XML deberíamos cambiar la llamada al Post:

ApiRest05Post

```
...
  // ***************
  // XML DEL LIBRO
  String libro = "<?xml version='1.0' encoding='UTF-8' standalone='yes'?>\n"+
                           "<libro>\n" +
                           "    <id>"+id+"</id>\n" +
                           "    <titulo>"+titulo+"</titulo>\n" +
                           "    <autor>"+autor+"</autor>\n" +
                           "</libro>  ";
  libro = libro.replaceAll("'", "\"");
  System.out.println(libro);

  // ***************
  // POST DEL LIBRO
  responsePost = client.target("http://localhost/fp/apirest")
                       .path("/libros/")
                       .request()
                       .header("Content-Type", "application/xml")
                       .accept(MediaType.APPLICATION_JSON)
                       .post(Entity.xml(libro));
...
```

Al usar los servicios de API Rest también podemos realizar **Mapping** a objetos.

En nuestro ejemplo, debemos crear la clase **Libros** y utilizarla al leer del objeto **response**.

Por ejemplo, partiendo del ejemplo **ApiRest02Get** tendríamos que indicarle a **readEntity** el tipo de Mapeo que queremos en vez de un **String.class**:

ApiRest06Get

```
...

// Mostrar la respuesta obtenida
System.out.println("Status:");
System.out.println("=======");
System.out.println(response.getStatus());
System.out.println();

System.out.println("Data Body:");
System.out.println("==========");

ArrayList<Libros> listaLib;
listaLib = response.readEntity(new GenericType<ArrayList<Libros>>() {});
System.out.println(listaLib);

System.out.println();

...
```

A la hora de realizar un Post, también cambiaríamos el **Entity**. Partiendo del ejemplo ApiRest04Post se podría cambiar creando el objeto libro en vez de el JSON:

ApiRest07Post

```
...
  // ***************
  // OBJETO DEL LIBRO
  Libros libro = new Libros(id, titulo, autor);
  System.out.println(libro);

  // ***************
  // POST DEL LIBRO
  responsePost = client.target("http://localhost/fp/apirest")
                       .path("/libros/")
                       .request()
                       .header("Content-Type", "application/json")
                       .accept(MediaType.APPLICATION_JSON)
                       .post(Entity.entity(libro, MediaType.APPLICATION_JSON));
...
```

Tanto **put** como **delete**, son métodos más sencillos de usar que get y post, ya que son variantes.

ANEXO – CONEXIÓN CON PROXY

Para poder utilizar una URL de internet utilizando el proxy, bastará con cambiar el ClientBuilder

```
Client client = ClientBuilder.newClient();
```

por otro con configuración:

```
ClientConfig config = new ClientConfig();
config.property(ClientProperties.PROXY_URI, "192.168.1.101:8080");
Client client;
client = ClientBuilder.newClient(config);
```

10. Componentes para servidores de aplicaciones

Ya hemos visto los tipos de aplicaciones (escritorio y web), algunos ejemplos de componentes de software como el de **Persona.jar** reutilizado en otras aplicaciones de escritorio y en aplicaciones web con JSP.

Plataformas Java

Hemos comprobado también que existe una equivalencia entre PHP y JSP para la creación de páginas web dinámicas, pero la realidad es que Java es mucho más potente.

Java tiene dos plataformas para ejecutar aplicaciones:

- Java J2SE (Java 2 Standard Edition)
- Java J2EE (Java 2 Enterprise Edition)

La plataforma **Java SE (Java Standard Edition)** es la base de la tecnología Java, incluyendo herramientas de desarrrollo, la Máquina Virtual Java (JVM) y la documentación para la programación.

Está orientada a la creación de aplicaciones cliente pero no incluye soporte par tecnologías de Internet.

Esto implica que son J2SE no podemos desarrollar aplicaciones web de Java.

La plataforma **Java EE (Java Enterprise Edition)** añade a Java la funcionalidad necesaria para convertirse en un lenguaje orientado al desarrollo de aplicaciones y servicios en Internet.

Con Java EE se pueden desarrollar sitios web complejos bajo la tecnología Java mediante la utilización de **JSP** (lenguaje de script de servidor para crear páginas web dinámicas como las de PHP a ASP) y **Servlets** (scripts CGI en el servidor similares a los creados con PERL)

Tipos de Componentes Java

Vemos ahora una primera clasificación de los componentes

JavaBean es un componente de software para la plataforma Java SE.

EJB (Enterprise JavaBean) es un componente de software para la plataforma Java EE que se despliega sobre un contenedor de EJB, incluido en un servidor de aplicaciones.

Si desplegamos una aplicación con componentes EJB en Apache Tomcat veremos que no funciona. (Lo veremos más adelante)

Apache Tomcat es un servidor web, desarrollado bajo el proyecto Jakarta en la Apache Software Foundation, con soporte de servlets y JSP (JavaServer Pages), pero no es un servidor de aplicaciones, por tanto, habría que incluirle módulos adicionales para ampliar sus posibilidades. Se usa como servidor web autónomo en entornos con alto nivel de tráfico y alta disponibilidad.

Como Apache Tomcat no es un Servidor Web de Aplicaciones Java necesitaremos usar otros servidores web que incorporen esta característica.

Servidor Web de Aplicaciones Java

Los dos más usados son:

WildFly, anteriormente **JBoss,** es un servidor de aplicaciones J2EE de código abierto implementado en Java. Ofrece una plataforma de alto rendimiento para aplicaciones de e-business.

Sus características principales son:

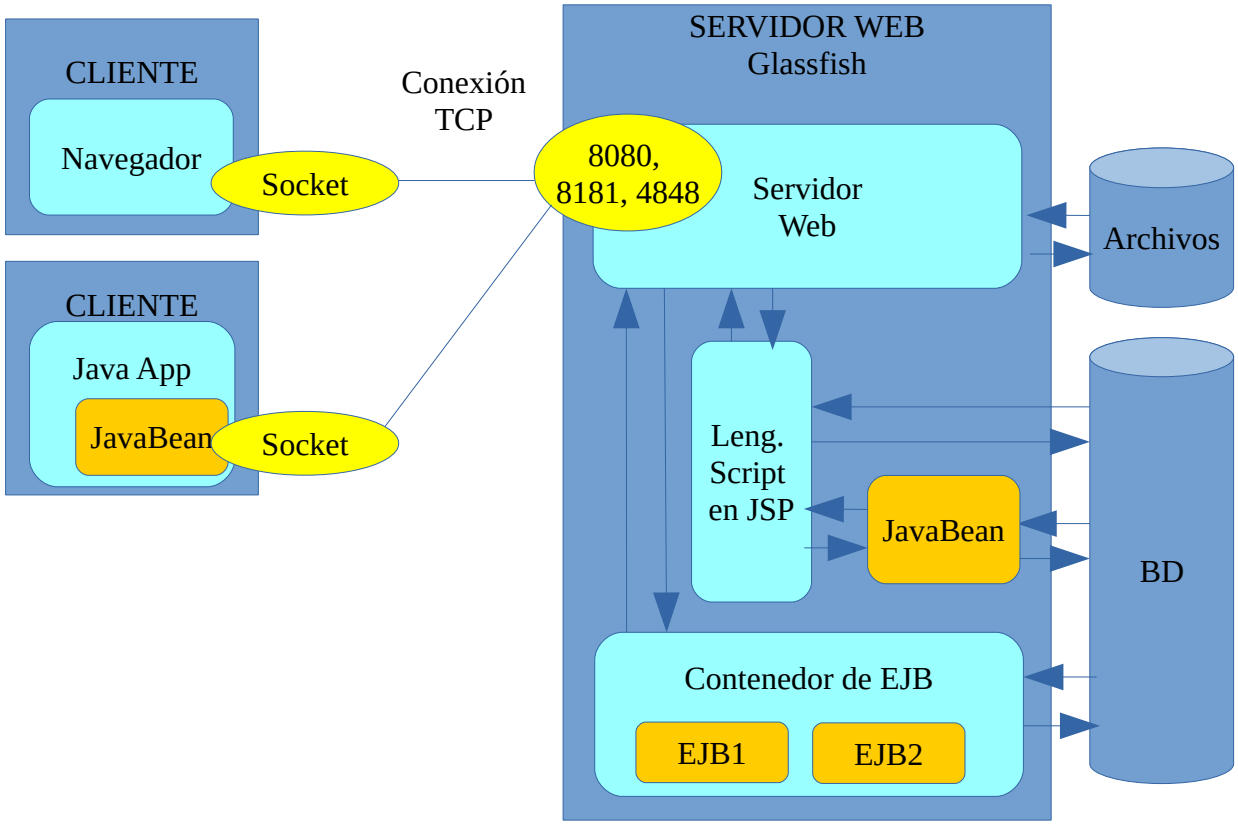
- Es distribuido bajo licencia de código abierto GPL/LGPL.
- Proporciona un nivel de confianza suficiente para ser utilizado en entornos empresariales.
- Es un servicio incrustable, por ello está orientado a la arquitectura en servicios.
- Servicio del middleware para objetos Java.
- Soporte completo para JMX (Java Management eXtensions).

GlassFish es un servidor de aplicaciones de código abierto que implementa funcionalidades de Java EE.

Es gratuito y de código abierto, desarrollado por Sun Microsystems y tiene como base al servidor Sun Java Application Server de Oracle Corporation, un derivado de Apache Tomcat.

Aunque en principio daría igual usar cualquiera de los dos, nosotros usaremos Glassfish porque tiene una instalación muy sencilla y se integra perfectamente en Netbeans, permitiéndonos ejecutar el debugger durante la ejecución.

De esta forma, el esquema para la programación en un servidor de aplicaciones Java como **Glassfish** quedaría:



Como podemos ver, el contenedor de aplicaciones EJB (Enterprise JavaBean) ejecutará los componentes desarrollados y generará una salida de datos para el servidor web.

11. Crear componentes para servidores de aplicaciones

Los servicios de API Rest se pueden implementar con diferentes tipos de tecnologías (hemos visto que se puede incluso con PHP), pero en los servidores de aplicaciones Java se suele implementar con componentes de software de tipo EJB lo que proporciona mayores prestaciones como mayor escalabilidad, mejor gestión de memoria, posibilidad de balanceo de carga, pool de conexiones, …

EAR (Enterprise Application aRchive) es un formato de archivo utilizado por Java EE para empaquetar uno o más módulos en un único archivo, de modo que la implementación de los distintos módulos en un servidor de aplicaciones se realice de manera simultánea y coherente. También contiene archivos XML llamados descriptores de implementación que describen cómo implementar los módulos.

11.1 Asistente Restful Web Services from Patterns

Para crear un proyecto de APIRest simple, sin JPA de acceso a datos, podemos usar el asistente "**Restful Web Services from Patterns**".

Vamos a crear un proyecto denominado **ApiRestHolaMundo** que proporcione un API Rest de Java que se instalará (deploy) en un Servidor de Aplicaciones de **Glassfish** con el asistente de NetBeans.

La aplicación tendrá una URL para mostrar el contenido JSP:

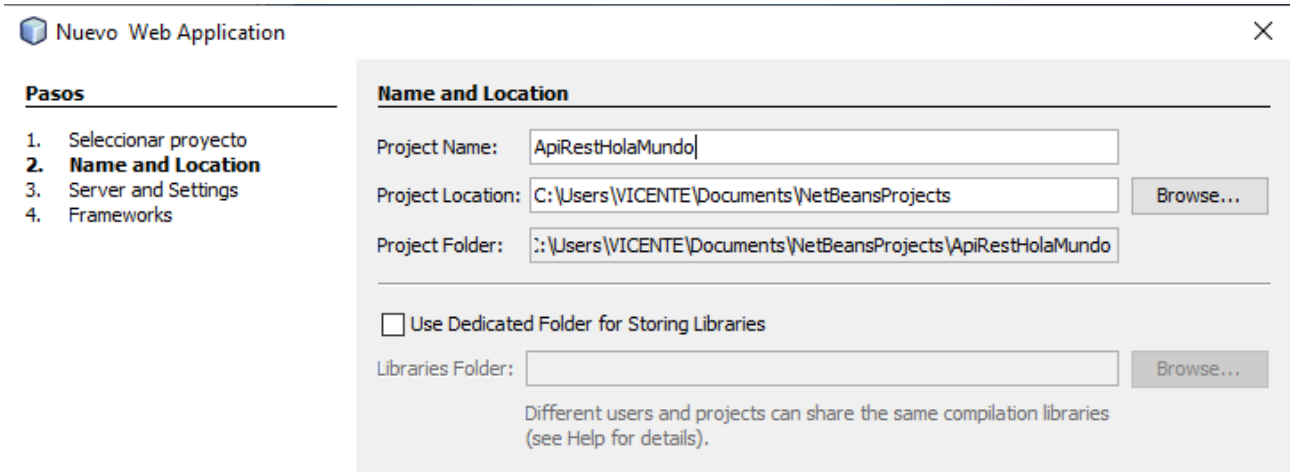
```
http://dominio/basico
```

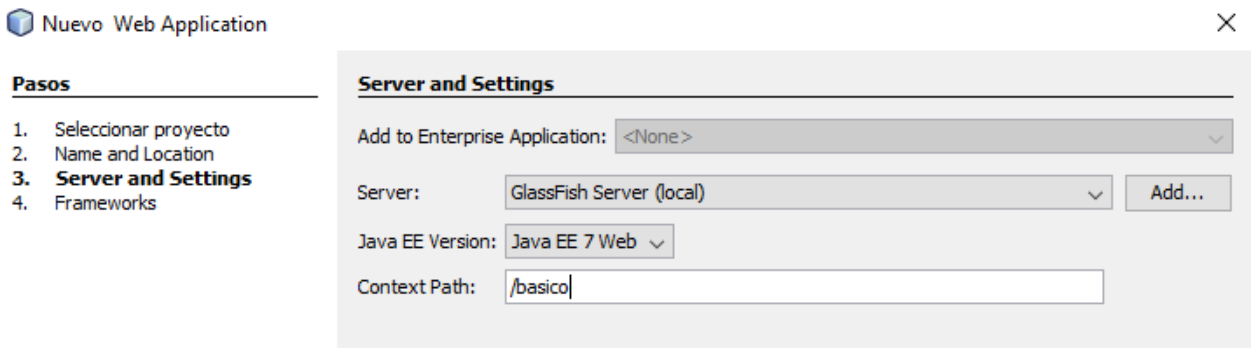
Y otra URL para activar el servicio API Rest:

```
http://dominio/basico/recursos/holamundo
```

Paso 1 – Crear proyecto de tipo Java Web

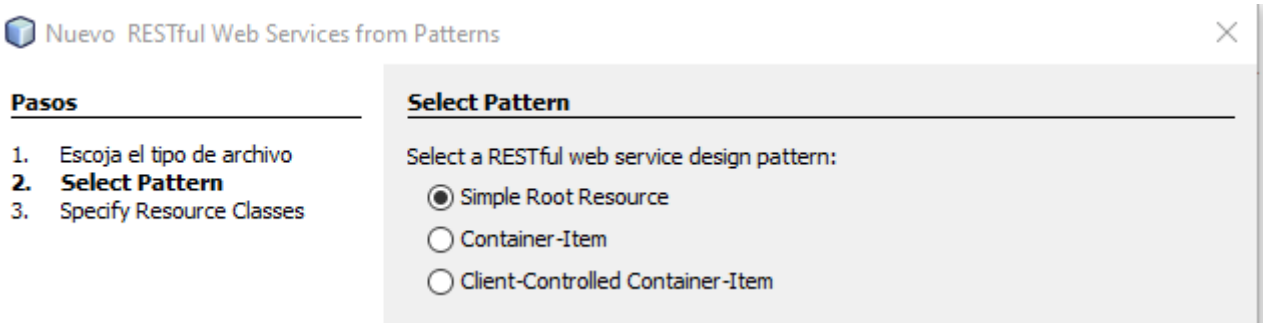
Utilizaremos el asistente:





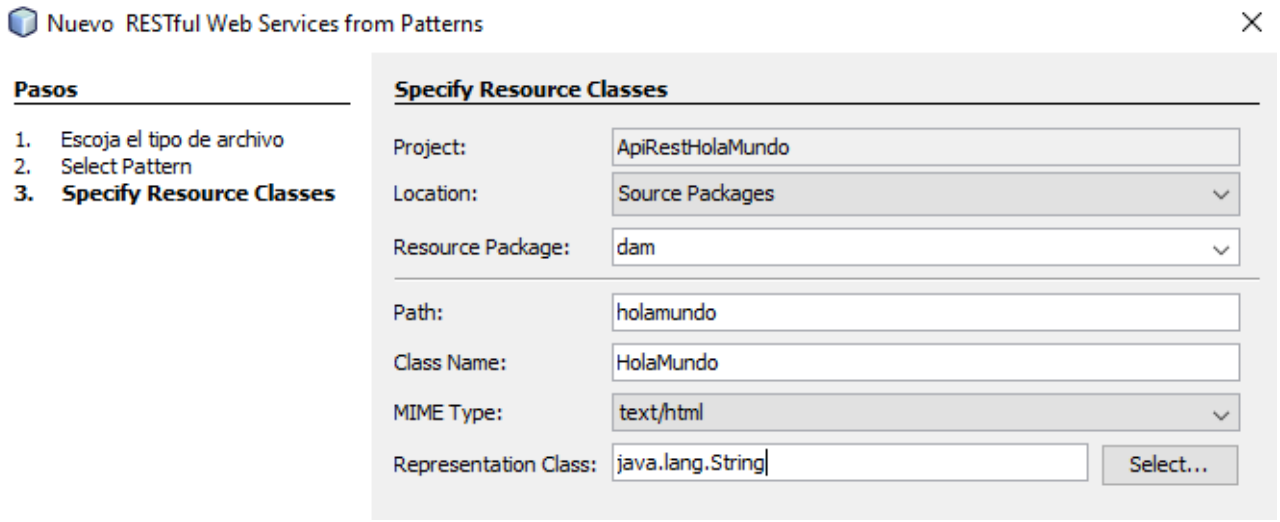
Paso 2 – Añadir archivo "Web Services"

Ahora añadiremos "Archivo nuevo" de la categoría "Web Services" del tipo de archivo "Restful Web Services from Patterns" con el patrón "Simple Root Resource".



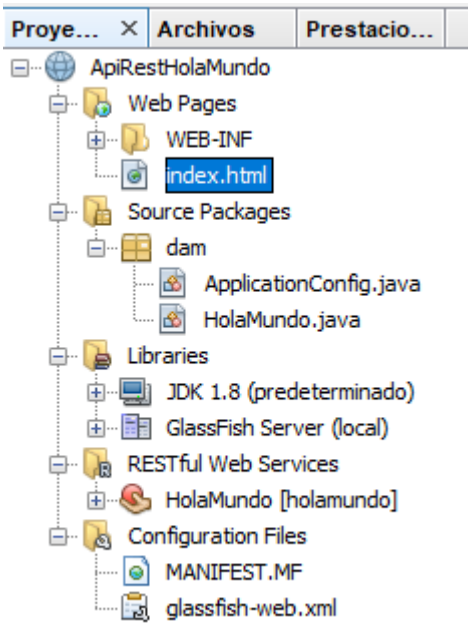
A continuación indicaremos:

- **Resource Package**: dam
- **Path**: holamundo
- **Class Name**: holamundo
- **MIME Type**: text/html
- **Representation Class**: java.lang.String



El asistente nos crea el siguiente proyecto donde:

ApplicationConfig.java
HolaMundo.java



Cuando lo ejecutemos, además tendremos:

RESTful Web Services => HolaMundo
Configuration Files

Veamos el contenido de los archivos principales:

ApplicationConfig.java	Prepara el servicio de API Rest añadiendo el **ApplicationPath** al **target** a la URL base. Por defecto es **webresources** pero puede cambiarse por **recursos**.
HolaMundo.java	Es el interfaz de métodos que se implementan y estarán disponibles para el servicio: GET

En HolaMundo.java elimnaremos el PUT y modificaremos el GET:

HolaMundo.java

```
...
    @GET
    @Produces(MediaType.TEXT_HTML)
    public String getHtml() {
       return "<html><body><h1>Hola Mundo!!</body></h1></html>";
    }
...
```

Ya está terminado y podemos ejecutarlo. El interfaz que nos ofrece es:

GET	**/recursos/holamundo**	Obtiene una texto en formato html

11.2 Creación de servicios sin acceso a datos

Para crear servicios REST utilizaremos anotaciones en los métodos:

@MÉTODO	Indica la operación que ejecutará el método: GET, POST, PUT DELETE.
@Consumes	Son los formatos que soporta el método en los datos recibidos
@Produces	Son los formatos que soporta el método en los datos enviados
@Path	Es el Path que se añade al Target en la URL
@PathParam	Es la anotación que asocia un parámetro del método a un elemento del Path

API Rest – Anotaciones
https://docs.oracle.com/cd/E19798-01/821-1841/6nmq2cp1v/index.html

Además, también usaremos datos de respuesta que se representan por unos códigos denominados "**Response Status Code**".

API Rest – Response.status
https://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Response.Status.html

ApiRestMath

Como ejemplo, podemos crear un proyecto denominado **ApiRestMath** que admita el método GET en la URL:

```
http://localhost:8081/math/apirest/aritmetica/sumar/3/5
```

El método tendrá la anotación:

```
@Path("/sumar/{operando1}/{operando2}")
```

11.3 Creación de servicios usando Bases de Datos

Para diseñar un servicio de API Rest con acceso a datos realizaremos varias clases:

1) **ApplicationConfig**: clase que define el path de la aplicación y añade las clases que van a responder como servicios en el API Rest
2) **ServiceREST**: clases que definen los métodos con su interfaz del servicio de API Rest
3) **DAO**: Objeto de Acceso a Datos que define las funciones que mapean los datos almacenados en las Bases de Datos con los objetos de clases de datos del API Rest

Un **Objeto de Acceso a Datos** (en inglés, **Data Access Object**, abreviado **DAO**) es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de datos o un archivo. El término se aplica frecuentemente al Patrón de diseño Object.

Los Objetos de Acceso a Datos son un Patrón de los subordinados de Diseño Core J2EE y considerados una buena práctica. La ventaja de usar objetos de acceso a datos es que cualquier objeto de negocio (aquel que contiene detalles específicos de operación o aplicación) no requiere conocimiento directo del destino final de la información que manipula.

Si seguimos la estructura, lo primero que deberemos desarrollar es la clase DAO de nuestro nuevo API Rest **DAOGimnasio**.

Paso 1) Crear un DAO

Veamos un **DAO** que utiliza **MongoDB** con una base de datos denominada **gimnasio** con dos colecciones llamadas **clientes** y **monitores** respectivamente.

En esta clase crearemos de los siguientes métodos:

DAOMongoGimnasio

```
public static boolean clientesExiste(Clientes cli)
public static boolean clientesPost(Clientes cli)
public static ArrayList<Clientes> clientesGetAll()
public static Clientes clientesGet(String dni)
public static void clientesPutTelefono(String dni, Clientes cli)

public static ArrayList<Monitores> monitoresGetAll()
public static Monitores monitoresGet(String idMonitor)
```

Estos son métodos que permitirán a **ServiceREST** acceder o obtener clientes y monitores de la Base de Datos.

Para crear y probar esta clase **DAOMongoGimnasio** puede crearse en un proyecto separado y probarla en un proyecto de tipo **Java Application** de los habituales.

DAOGimnasio
Crear un proyecto denominado **DAOGimnasio** que acceda a datos de las tablas clientes y monitores de la BD de MongoDB denominada gimnasio con los métodos definidos en **DAOMongoGimnasio.**

Paso 2) Crear ServiceREST

ApiRestMongoDB
Crear un proyecto **API Rest** denominado **ApiRestMongoDB** que acceda a datos de las tablas clientes y monitores de la BD de MongoDB denominada gimnasio utilizando las clases definidas en el proyecto **DAOGimnasio.**

Crearemos dos **ServiceRest**, uno para Clientes y otro para Monitores.

ServiceRESTClientes		
GET	**/clientes**	**Obtiene todos los clientes**
GET	**/clientes/{dni}**	**Obtiene un cliente**
POST	**/clientes**	**Inserta un cliente nuevo**

ServiceRESTMonitores		
GET	**/monitores**	**Obtiene todos los monitores**
GET	**/monitores/{monitor}**	**Obtiene un monitor**

Paso 3) Configurar ApplicationConfig

Estableceremos el path de la aplicación a **recursos** y añadiremos los dos **ServiceRest** desarrollados.

12. Uso de JPA para componentes de servidores de aplicaciones

Si trabajamos con un Servidor de Base de Datos Relacional como MySQL, necesitaremos realizar un mapping entre las tablas/registro de MySQL y los objetos de Java.

Este proceso se realiza habitualmente con la JPA.

Java Persistence API, más conocida por sus siglas **JPA**, es la API de persistencia desarrollada para la plataforma Java EE. Es un framework del lenguaje de programación Java que maneja datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard y Enterprise

Ya vimos anteriormente que el ORM de Hibernate usa JPA para realizar el mapeo.

En API Rest también suele usarse JPA.

12.1 Asistente Restful Web Services from Database

ApiRestLibros

Vamos a crear un proyecto denominado **ApiRestLibros** que proporcione un API Rest de Java que se instalará (deploy) en un Servidor de Aplicaciones de **Glassfish** con el asistente de NetBeans.

La aplicación tendrá una URL para mostrar el contenido JSP:

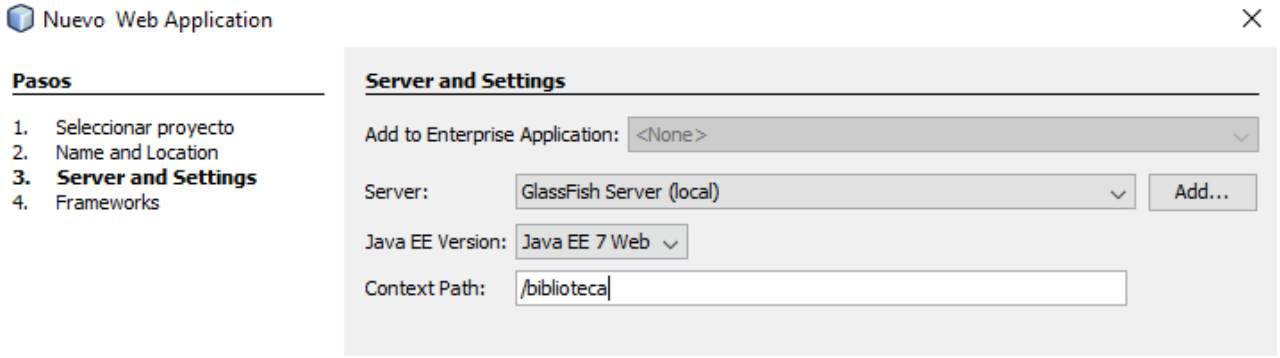
`http://dominio/biblioteca`

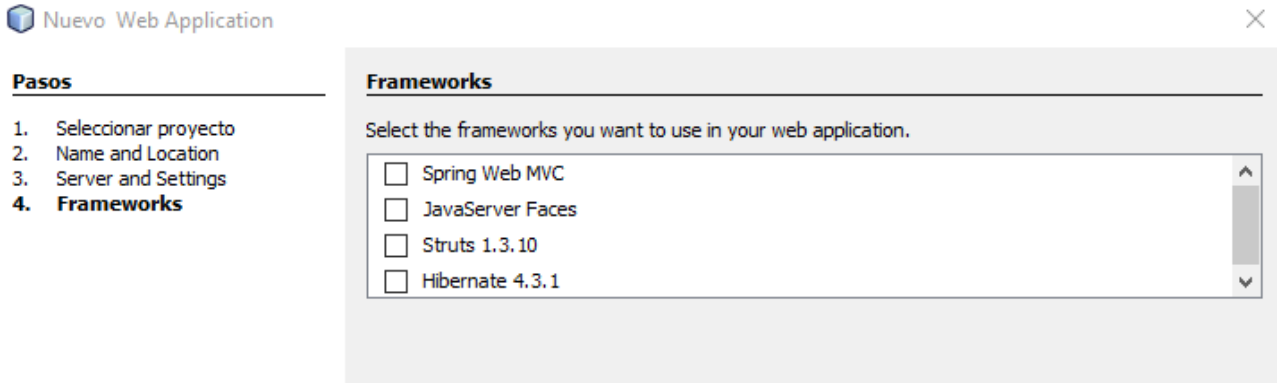
Y otra URL para activar el servicio API Rest:

`http://dominio/biblioteca/recursos/dam.libros`

Paso 1 – Crear proyecto de tipo Java Web

Utilizaremos el asistente:

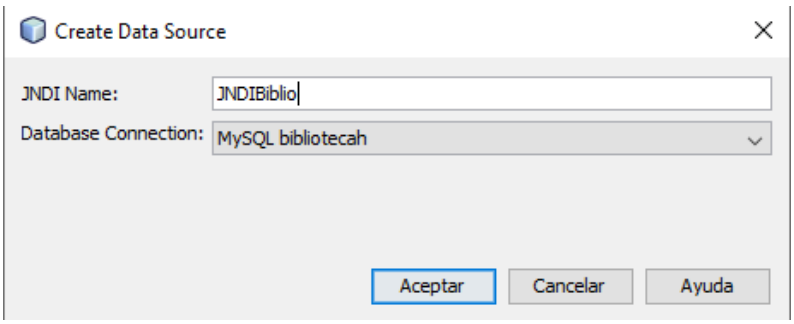




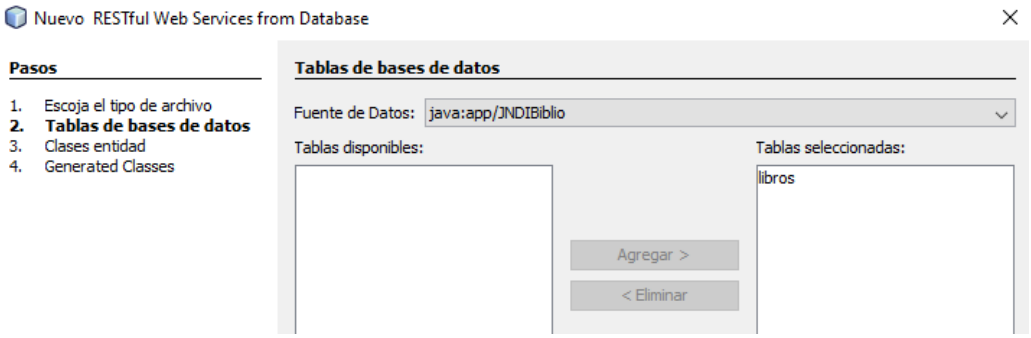
Paso 2 – Añadir archivo "Web Services"

Ahora añadiremos "Archivo nuevo" de la categoría "Web Services" y el tipo de archivo **"Restful Web Services from Database"**

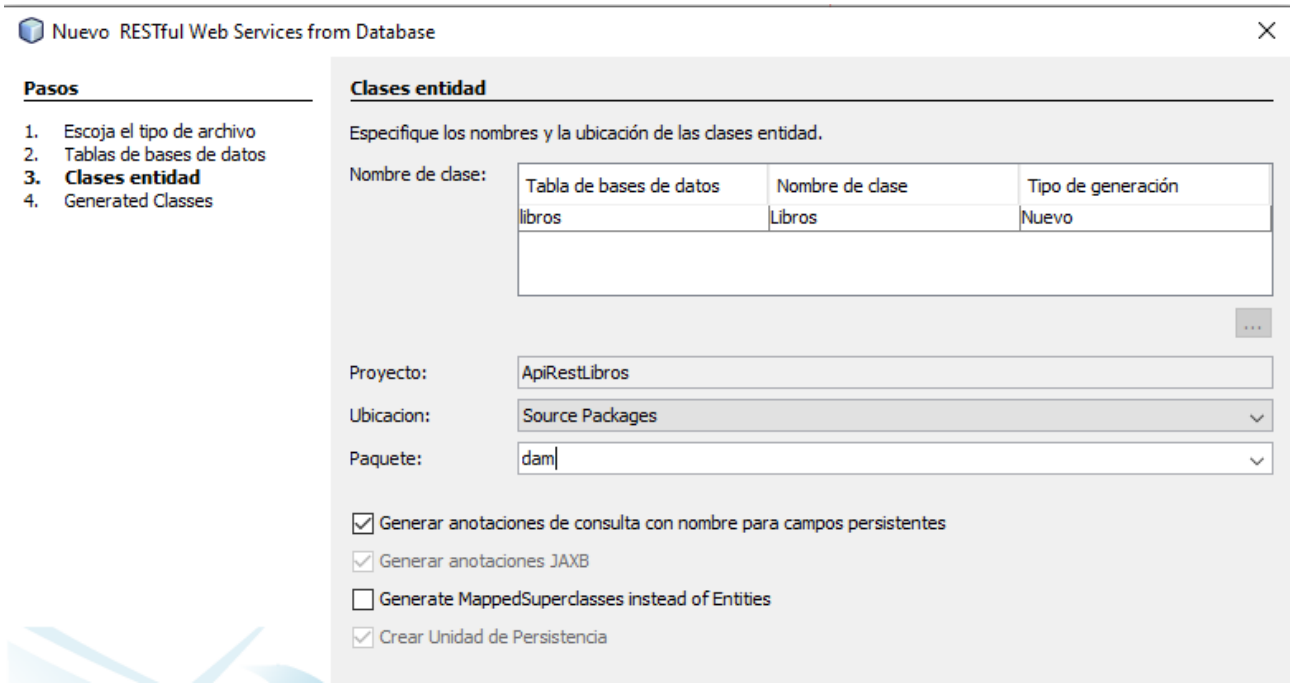
En Fuente de Datos añadiremos un nuevo **"Data Source"**.



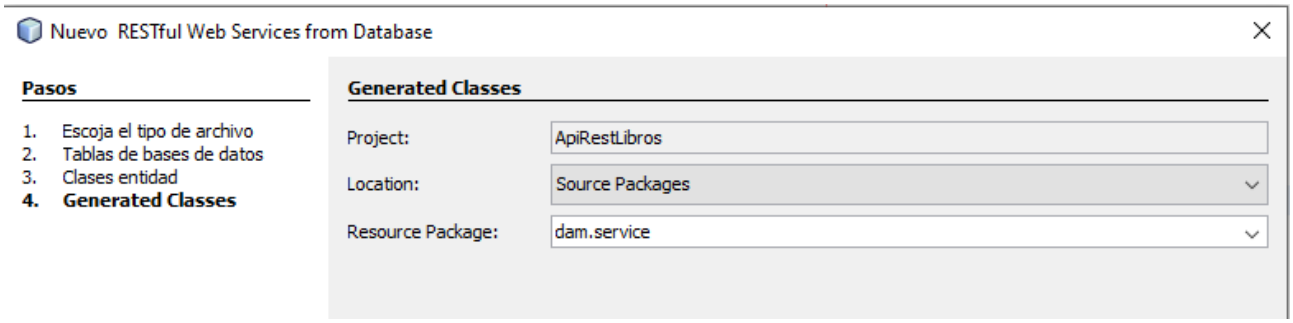
Una vez cargada, podremos seleccionar la tabla libros:



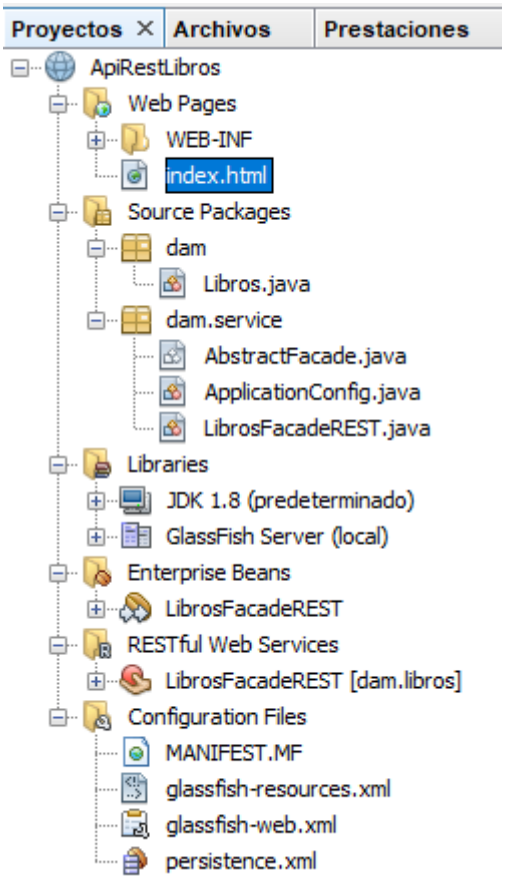
Generaremos un añadido en el path para la tabla denominado **dam.**



Y ahora se generará el Servicio:



El asistente nos crea el siguiente proyecto donde:



Libros.java
AbstractFacade.java
ApplicationConfig.java
LibrosFacadeREST.java

Además tenemos:

Enterprise Beans => LibrosFacadeREST
RESTful Web Services => LibrosFacadeREST
Configuration Files

Veamos el contenido de los 4 archivos principales:

Libros.java	Es la clase Libros para mapear la tabla libros
ApplicationConfig.java	Prepara el servicio de API Rest añadiendo el **ApplicationPath** al **target** a la URL base. Por defecto es **webresources** pero puede cambiarse por **recursos**.
LibrosFacadeREST.java	Es el interfaz de métodos que se implementan y estarán disponibles para el servicio: GET, POST, PUT, DELETE. Además contiene el **path** que se añade al **target**.
AbstractFacade.java	Es una clase abstracta que contiene las funciones que usando el **getEntityManager** del mapeo realizado por **javax.persistence** del **JPA,** serán ejecutados por los métodos desde **LibrosFacadeREST,**

Ya está terminado y podemos ejecutarlo. El interfaz que nos ofrece es:

GET	`/recursos/dam.libros`	Obtiene una lista con todos los registros (XML, JSON)
GET	`/recursos/dam.libros/{id}`	Obtiene un registro (XML, JSON)
GET	`/recursos/dam.libros/{from}/{to}`	Obtiene una lista con algunos los registros (XML, JSON)
GET	`/recursos/count`	Obtiene el número de registros (TEXT)
POST	`/recursos/dam.libros`	Inserta un registro (XML, JSON)
PUT	`/recursos/dam.libros/{id}`	Actualiza un registro (XML, JSON)
DELETE	`/recursos/dam.libros/{id}`	Elimina un registro

Ahora podemos eliminar, modificar o añadir métodos a **LibrosFacadeREST,** así como nueva funcionalidad a **AbstractFacade.**

Por ejemplo, vamos a añadir una nueva funcionalidad al servicio API Rest que nos permita **recuperar el primer registro de libros**. Para ello habrá que realizar dos cambios:

Funcionalidad

Añadir a **AbstractFacade** un nuevo método que devuelva una lista de Libros con un sólo registro, el primero. Este método se llamará findFirst:

AbstractFacade

```
...
public List<T> findFirstList() {

  javax.persistence.criteria.CriteriaQuery cq =
                      getEntityManager().getCriteriaBuilder().createQuery();
  cq.select(cq.from(entityClass));
  javax.persistence.Query q = getEntityManager().createQuery(cq);

  q.setMaxResults(1);       // Cantidad de registros de la lista
  q.setFirstResult(0);      // Comenzando en elitem 0 de la lista de resultados

  return q.getResultList(); // Devolvemos una lista de objetos

}
...
```

Interfaz del servicio

Añadir a **LibrosFacadeREST** un nuevo método que llame a nuestro nuevo método de funcionalidad e indique el path al servicio **/dam.libros/firstlist**:

LibrosFacadeREST

```
...
@GET
@Path("firstlist")
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public List<Libros> firstlistREST() {
    return super.findFirstList();
}
...
```

Para poder proporcionar más funcionalidad, debemos conocer el **javax.persistence** de **JPA**:

API Rest – Status Code

https://www.objectdb.com/api/java/jpa/Query/getSingleResult